Towards a Portable and Future-proof Particle-in-Cell Plasma Physics Code

R. F. Bird, S. J. Pennycook, S. A. Wright, S. A. Jarvis
Performance Computing and Visualisation
Department of Computer Science
University of Warwick
rfb@dcs.warwick.ac.uk

ABSTRACT

We present the first reported OpenCL implementation of EPOCH3D, an extensible particle-in-cell plasma physics code developed at the University of Warwick. We document the challenges and successes of this porting effort, and compare the performance of our implementation executing on a wide variety of hardware from multiple vendors. The focus of our work is on understanding the suitability of existing algorithms for future accelerator-based architectures, and identifying the changes necessary to achieve performance portability for particle-in-cell plasma physics codes.

We achieve good levels of performance with limited changes to the algorithmic behaviour of the code. However, our results suggest that a fundamental change to EPOCH3D's current accumulation step (and its dependency on atomic operations) is necessary in order to fully utilise the massive levels of parallelism supported by emerging parallel architectures.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*Parallel Programming*

General Terms

Experimentation, Performance

Keywords

OpenCL, Particle-in-Cell, Performance Portability, Optimisation

1. INTRODUCTION

The plasma physics applications required by many high-performance computing (HPC) centres are expensive to produce, due to the highly complex nature of the mathematics involved and the desired ability to support a wide range of simulations within a single software package. Developing such an application (a process which can be spread over many years, or even decades) also represents a significant investment, often including the creation of novel algorithms that must be rigorously tested by domain experts and computer scientists before the application can be deployed into a production environment. It is therefore important to such centres that codes remain usable for as long as possible, to maximise the return on their investment.

Many such codes (including the one featured in this study) were written to target traditional HPC clusters comprised of a large number of high-clocked serial cores connected via some networking interface. As such they have been engineered to ensure efficient inter-node message passing patterns and scaling behaviours. but are not likely to be well-prepared for the many forms of on-node parallelism that typify the most recent and future architecture designs [6]. It is therefore crucial that codes are revisited to ensure both: 1) portability between the different architectures available today; and 2) continued scalability on the architectures of tomorrow.

Since it is not feasible to rewrite an entire production code from scratch for each new technology, it is important to understand how to introduce the desired portability and future-proofing in an incremental manner, and to do so without becoming tied to a proprietary language or programming interface. To this end, we present a case-study detailing the porting of a production plasma physics code to accelerator architectures using the Open Computing Language (OpenCL) [10], one of many available open standards for targeting accelerators [16, 17, 18].

In this paper we aim to develop an understanding of techniques that allow legacy Fortran codes (that are typically serial in nature), and in particular particle-in-cell (PIC) plasma physics applications, to benefit from the increasing amounts of on-node parallelism present in emerging HPC architectures. We explore the utility of OpenCL for future-proofing a production application, and the extent to which accelerator hardware can lead to application speed-ups in the PIC domain. The eventual goal of this work is to feed any identified code optimisations back into the original code base; such improvements may have been overlooked for several years, and may therefore be responsible for many wasted compute cycles on large HPC systems.

The specific contributions of this work are:

- We present the first documented port of EPOCH3D to accelerator architectures. Due to the size and complexity of its code base, making the OpenCL implementation of EPOCH3D fully resident on the accelerator (*i.e.* running all computation on the device, to minimise the frequency of data transfers across the PCIe bus) is an ambitious undertaking and beyond the scope of this work; we focus instead on the porting of only its most expensive subroutine (`push_particles`). EPOCH3D is widely used by both the academic community and by industry, and we believe that many of the issues we experience are representative of those one would expect to encounter in any legacy Fortran application.

- We explore optimisation opportunities within the OpenCL code, with an aim to improve the performance of EPOCH3D running on accelerated systems. Specifically, we examine the utility of: 1) SIMD/SIMT execution on CPUs/GPUs; 2) loop and kernel fission; and 3) grouping the computation of particles within a given cell. We demonstrate the existence of a trade-off between optimising the field calculation and current accumulation steps of EPOCH3D, and highlight the difficulty of mapping EPOCH3D's current accumulation step to accelerator architectures.

- Finally, we present a detailed benchmarking comparison of our OpenCL port executing on a wide variety of different hardware and architecture types. In addition to reporting performance for a number of accelerator architectures – three generations of NVIDIA hardware (Tesla, Fermi and Kepler) – we also compare the performance of the original Fortran code to the OpenCL implementation running on traditional CPU architectures. Our results suggest that a fundamental change to EPOCH3D's current accumulation step will be necessary in order to fully utilise emerging parallel architectures.

The remainder of this paper is organised as follows: Section 2 provides a survey of related work; Section 3 provides background information about PIC applications and the specific algorithms used in EPOCH3D; Section 4 details the development of our OpenCL implementation of EPOCH3D; Section 5 presents a comparative benchmarking study, examining the performance of both individual kernels and the application as a whole; and finally Section 6 concludes this paper.

2. RELATED WORK

The use of emerging parallel architectures for the acceleration of scientific codes is well documented, with many researchers reporting speed-ups of up to an order of magnitude following optimisation [2, 15, 22, 25]. Others have highlighted the importance of ensuring fair comparisons between different hardware types [3, 4, 13, 24]; to avoid artificially inflating any claims about the performance of our accelerator-based solutions we include all known overheads (such as PCIe transfers) and acknowledge it is important to consider the impact of kernel-level speed-ups in the context of overall application time.

The utility of GPU architectures in the PIC domain has been explored in previous work. In [9], the authors discuss the importance of carefully partitioning the grid space, suggesting that an efficient GPU PIC solution requires fine tuning and extended programmer effort. They also discuss the importance of efficient use of shared memory on GPUs – this feature is not available on x86-based platforms (*e.g.* CPUs and Intel Xeon Phi coprocessors), and may therefore negatively impact the ability to offer portable performance. In [23] the authors discuss a range of algorithms to perform grid interpolation, and discuss a range of potential issues including shared memory and thread contention; and in [5], PIC solutions have been shown to scale across multiple accelerators while maintaining numerical stability.

The extensible nature of EPOCH3D constrains the set of allowable algorithmic implementations, limiting the extent to which we are able to change the core algorithm, a consideration not held by other PIC applications. An example of this is that EPOCH3D's algorithm is required to ensure that the electric fields satisfy Poisson's equation, a condition that many non-relativistic codes do not have to meet. In this paper, we seek to determine whether these algorithms can benefit from the levels of parallelism present in accelerator architectures, or whether it will be necessary to adopt a fundamentally different approach.

Open standards (*e.g.* OpenMP [17], OpenCL [10] and OpenACC [16]) provide application developers a greater level of portability than third-party libraries and languages, since they are supported by multiple compilers and on multiple architectures. Such standards potentially also provide some level of future-proofing, with each updated specification introducing new constructs required to exploit recent architectural changes (*e.g.* OpenMP is adding support for the vectorisation of loops [11] and offloading computation to accelerators [18]). However, a common criticism of these standards is that although they guarantee *functional* portability, they make no guarantees of *performance* portability (*i.e.* the ability of a single source code to achieve good levels of performance on a wide variety of hardware). The development of performance-portable OpenCL codes has been the subject of previous work [7, 12]; we build upon our own prior research, in which we utilised OpenCL for the acceleration of wavefront [19] and molecular dynamics [20] codes.

3. BACKGROUND

EPOCH3D [1] is a production code developed at the University of Warwick as part of the Extendable PIC Open Collaboration project, which aims to write and maintain a UK community advanced relativistic electromagnetism PIC code. The core algorithm of EPOCH3D is based upon the particle push and field update algorithms developed by Hartmut Ruhl [21], extended to include advanced features such as collisions, ionisation and quantum electrodynamics (QED) driven coherent radiation. EPOCH3D tracks the electric and magnetic fields generated by the motion of pseudo-particles as they move across a grid, and is therefore capable of reproducing the full range of classical microscale behaviour of a collection of charged particles.

```
for all species do
    for all particles do

        ▷ Move particles.
        position ← position + momentum

        ▷ Update momentum based on field effects.
        e_cell ← ⌊position⌋
        for all neighbours of e_cell do
            calculate electric field effects
        end for

        b_cell ← ⌊position + 0.5⌋
        for all neighbours of b_cell do
            calculate magnetic field effects
        end for

        momentum ← momentum +
                        electric and magnetic field effects

        ▷ Calculate and deposit currents.
        for all neighbour cells do
            calculate current
            deposit current
        end for

    end for
end for
```

Figure 1: Pseudo-code for `push_particles`.

The core of EPOCH3D's PIC algorithm (the `push_particles` subroutine, given as pseudo-code in Figure 1) is the single biggest contributor to its execution time ($> 70\%$). It consists of three distinct steps, which are applied to every particle in turn:

1. Move the particle in space (according to its momentum), potentially crossing into one of many neighbouring cells.
2. Update the particle's momentum based upon the electric and magnetic fields.
3. Deposit current onto the grid due to the motion of the charged particle.

This algorithm features a large amount of inherent parallelism: the calculations for the position and momentum of each particle are independent, and the electric and magnetic fields remain constant between loop iterations. However, the current accumulation step contains a write-conflict; any number of particles (which are close in space) may attempt to update the charge for the same grid point. This conflict prevents the loop from being parallelised in a naïve fashion (*e.g.* via the insertion of pragmas, or source-to-source translation), and necessitates some form of global synchronisation between work-items, in order to ensure that no two work-items can update the same cell simultaneously.

When EPOCH3D (and many similar codes) were originally developed, such factors did not need consideration – during single threaded scalar execution, the grid updates of multiple particles cannot conflict with one another. As hardware designs have become increasingly focused on parallel execution, inherently serial algorithms such as this will become more of a performance bottleneck. Indeed, the issues we encounter with EPOCH3D's current accumulation step are not specific to OpenCL, or to accelerators: the write-conflict complicates the use of parallelism on CPU and GPU architectures alike.

4. OPENCL IMPLEMENTATION

In the original EPOCH3D code base, and as shown in Figure 1, the `push_particles` subroutine consists of a deep loop nest spanning around 700 lines of code. This is reasonably typical of large legacy codes that have been optimised for serial execution, exhibiting several properties that map well to traditional CPU architectures – in EPOCH3D's case, particle data are stored in a linked list of structs, and read/written only once in the entire loop, in order to improve cache performance. Such optimisations may actually be detrimental to performance on modern, parallel architectures: large kernels typically require many registers, and may cause register spilling; grouping dependent and independent operations in a single loop may completely prevent parallelisation (as explained in Section 3); and struct-of-arrays (SoA) data layouts are better suited to SIMD execution.

To address these concerns, we *fission* our OpenCL kernel into three parts, corresponding to the three operations discussed previously: 1) moving the particles; 2) updating particle momenta; and 3) updating the current. The overhead of kernel fission in this case is relatively low, requiring only a small number of values to be recalculated between kernels, and allows us to apply different optimisations to the three different kernels (which we will show to be important). It also separates the update conflict from the vast majority of the compute, and the same fissioning process could therefore enable the use of auto-vectorisation/OpenMP on CPUs – we intend to explore this in future work.

4.1 Particle Move

After fission, the particle move step is simple to represent in OpenCL. We assign a single particle to each OpenCL work-item, and allow the OpenCL runtime to decide upon the best work-group size for a given device. Since we make no use of shared memory, or other architecture-specific features, we expect this kernel to exhibit portable performance across hardware types. The kernel is memory bound – it simply reads in particle positions and momenta, performs a small number of floating-point operations, and then writes out the new positions – and we would therefore expect its performance to scale with memory bandwidth.

The only optimisations employed in this kernel concern the original particle data layout used by EPOCH3D. Whereas the original Fortran code uses a linked list of particle structs, we store particle data in an SoA layout and thereby ensure that memory accesses to particle data are coalesced.

4.2 Field Calculation

The second step of the particle push, in which particle momenta are updated based on the surrounding electric and magnetic fields, is also data-parallel following fission. However, unlike the particle move step, there are many optimisation opportunities. Each particle must read field information from the 27 cells surrounding its own (*i.e.* a 27-point stencil), and the optimisation of such kernels for GPU architectures is well-studied [14].

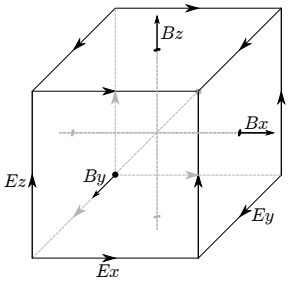


Figure 2: An example of a Yee staggered grid.

In the original Fortran implementation of EPOCH3D, particles are not stored in any particular order. Therefore, these stencil operations additionally become *gather* operations (*i.e.* uncoalesced loads); a collection of W particles (assigned to W work-items), may lie in W different cells and access $27W$ memory locations.

In order to improve this memory access pattern, we exploit domain-specific knowledge to reorder the particles before computing their momenta. Specifically, particles are sorted using a simple binning kernel, based upon a combination of their position and half-cell shifted position (to account for the Yee staggered grid demonstrated in Figure 2).

Although sorting in this way increases the probability that W particles will read from the same cells, it does not guarantee it; this prevents us from exploring other optimisations (such as using shared memory to cache cell data). In order to address this issue, we consider three alternative levels of parallelism: 1) assigning one work-item per particle (the default); 2) assigning one work-group per cell; and 3) assigning one work-group to some set of cells (henceforth referred to as a "super-cell").

Each of these three levels of parallelism maps intuitively to different distributions of particles: 1) a very sparse distribution, with a small number of particles per cell; 2) a very dense distribution, with many particles per cell; and 3) a distribution somewhere in between, where the size of the super-cell is set so as to contain a specific number of particles. In cases 2) and 3), shared memory can be used to reduce the number of accesses to global memory.

4.3 Current Accumulation

In order to overcome the write-conflict present in the current accumulation step of the particle push, we must introduce some form of global synchronisation between work-items to ensure that no two work-items can update the same memory location simultaneously.

There are a number of well-known approaches for this kind of mutual exclusion: critical sections; cell-wise (or super-cell-wise) locking; and atomic operations. However, implementing any of these approaches in a portable manner is challenging: different hardware has different levels of support for atomics (*e.g.* 32-bit, but not 64-bit, integers); and all three depend on extensive use of OpenCL's `atom_cmpxchg` routine, careless use of which can lead to deadlocks on some hardware (*e.g.* NVIDIA GPUs, where work-items are scheduled for execution in lock-step).

	CPUs		GPUs		
	X5550	**E5-2670**	**C1060**	**C2050**	**K20**
Manufacturer	Intel	Intel	NVIDIA	NVIDIA	NVIDIA
Compute Units	4	8	30	14	14
Proc. Elements	4	8	240	448	2496
Peak[a] GFLOP/s	85.12	332.8	933	1288	3520
Bandwidth (GB/s)	32	51.2	102	144	208
Power (Watts)	95	115	189	238	225
Compiler and Flags	Intel 13, `-O3`				
MPI Library	IMPI 4.1.0.030				
OpenCL SDKs	AMD APP 2.8		CUDA Toolkit 5.0		

[a]Peak performance for single precision.

Table 1: Hardware and software configuration.

A further complication influencing the performance of atomic locks is data access pattern. If two work-items attempt to update the same cell simultaneously, then one must wait until the other has finished – performance can therefore be increased by maximising the temporal distance between two accesses to the same memory location, in order to avoid contention between work-items. The end result is a performance trade-off between the field calculation and current accumulation kernels: the best case for one is the worst for the other. This is discussed further (alongside full performance results for the different approaches) in Section 5.

As an alternative to the current implementation, it is possible to replace EPOCH3D's current accumulation step with one more suitable for accelerator architectures. For example, one approach referred to as "classic PIC" does not accumulate current densities at all, instead calculating the current density in a later step by summing over particles. However, this leads to the electric fields no longer satisfying Poisson's equation, and requires a corrector step which distorts the electromagnetic wave dispersion relation – this scheme is unsuitable for relativistic problems. We are currently working with physicists at the University of Warwick to develop a new scheme, which is data-parallel and exhibits the desired physical characteristics, and hope to report on its performance in future work.

5. RESULTS

5.1 Experimental Setup

The hardware and software configuration for all experiments is given in Table 1. The reader's attention is drawn to the presence of two CPUs: we use the X5550 (which is based on the Nehalem microarchitecture) to compare the performance of the original Fortran code to that of our OpenCL implementation; and we use two E5-2670 sockets (based on the newer Sandy Bridge microarchitecture) as the baseline for comparisons between CPU and GPU architectures. The OpenCL runtime was not available to us on the Sandy Bridge system.

For the following experiments, we use EPOCH3D to simulate the interaction of two densely packed electron streams, with runtime options typical of normal conditions: each particle is assumed to have an individual mass; we use the (default) triangular pseudo-particle shape function; and we run a modest problem size of 25 million particles (128 particles per cell, for a 58^3 cell problem), with periodic boundary conditions. The CPU code was setup to issue manual prefetches

Approach	X5550	C1060	C2050	K20
Crit. Section	0.003	–	0.260	0.855
Lock (Face)	0.001	6.118	0.054	0.088
Lock (Row)	0.001	0.396	0.016	0.011
Lock (Cell)	0.002	0.041	0.011	0.003
Atomic Add	0.002	0.026	0.008	0.002

Table 2: Comparison of execution times (in seconds) for different mutual exclusion approaches.

between loop iterations (on account of particles being stored in a linked list), and we employ loop fission to facilitate a more accurate comparison between the three different kernel components. This fissioned version of the loop is ≈5–10% slower than the original Fortran code, due to the introduction of additional memory accesses.

5.2 Effects and Portability of Optimisations

5.2.1 Mutual Exclusion

Table 2 compares the performance of three alternative mutual exclusion approaches (critical sections, locking, and atomic adds), used to prevent write-conflicts within the current accumulation kernel. We report execution times for multiple granularities of locking: locking a face (*i.e.* using one lock per z co-ordinate); locking a row (*i.e.* using one lock per pair of y and z co-ordinates); and locking individual cells. All locks make use of a 32-bit `atomic_cmpxchg` operation.

There are several interesting trends in the data. On the CPU, we see that although using locks or atomic adds is faster than using a critical section, the locking granularity has little effect on kernel performance. This is due to the low number of work-items (relative to GPU hardware) that are able to contend for any given lock. On the GPUs, performance improves as the mutual exclusion becomes finer, with atomic adds providing the best runtime in all cases; this is a somewhat surprising result, since we had expected the overhead of acquiring locks (via atomic operations) to be more expensive than the writes to memory. Also of interest is that NVIDIA's newest Kepler architecture is out-performed by the older Fermi architecture in the presence of coarse locks – this is likely to be a reflection of the greater levels of parallelism present in Kepler, and the increased lock contention this may cause.

To investigate the overhead of atomic adds on different architectures, we repeat our experiments using normal writes (allowing the kernel to write to conflicting memory addresses, and achieve the wrong answer). Although we appreciate that this baseline is unrealistic, we use it here merely to represent a "best case" for writing to global memory. Our results (Figure 3) show that the overhead of atomics is inconsistent across different hardware: 2× on the CPU; 14× on the Tesla; 16× on the Fermi; and 2× on the Kepler. That the performance of atomics have been so greatly improved in the Kepler architecture is promising, but a 2× overhead is still undesirable for such an expensive kernel.

Since atomic adds provide the best performance on the architectures considered here, we use this version of the current accumulation kernel in all of the experiments that follow. However, we stress that this kernel is *not* performance

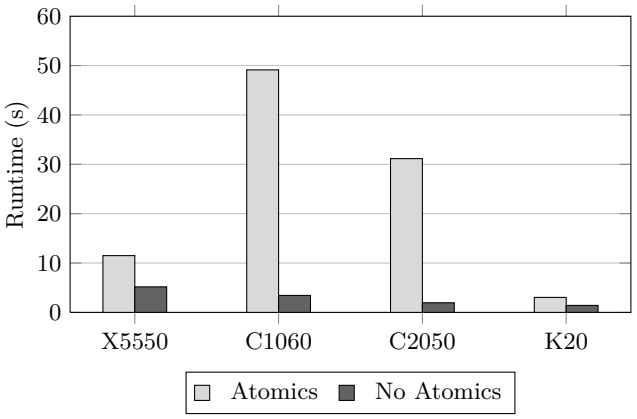


Figure 3: Overhead introduced by atomic operations.

portable since, as noted previously, some OpenCL runtime platforms do not feature support for 64-bit atomics. This is an issue in software, rather than hardware – at the time of writing, it is not possible to run our kernel on the X5550 processor using Intel's runtime, but it is possible using AMD's. Combined with the varying overhead of atomics on different architectures (and the inherent serialisation that will arise as parallelism, and hence lock contention, increases) there is clear motivation for further investigation of alternative algorithms in future work.

5.2.2 Particle Ordering

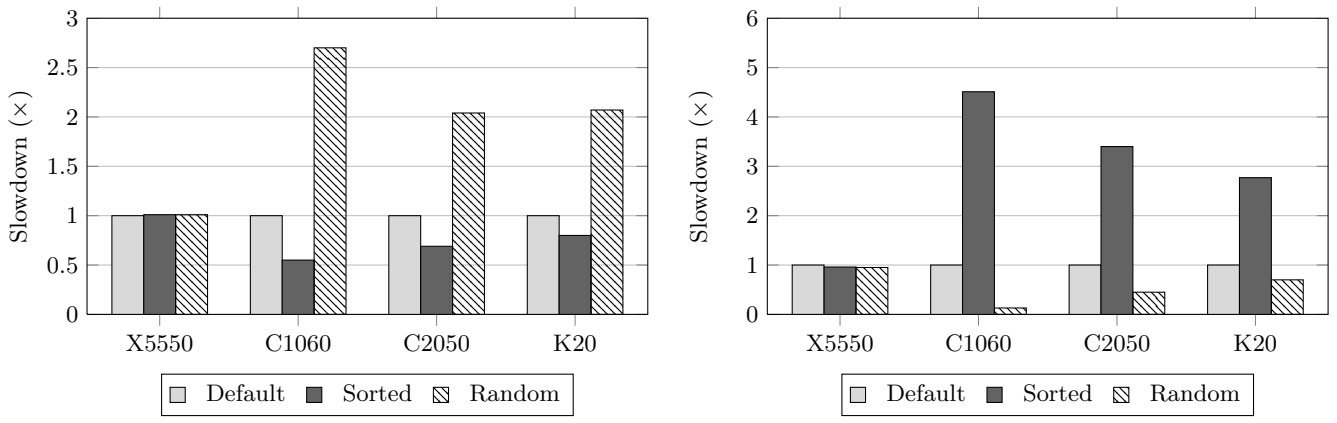
The graphs in Figure 4 present runtimes for (a) the field calculation kernel; and (b) the current accumulation kernel, using atomic adds. We compare three different particle orderings: the default ordering used by EPOCH3D (Default); an ordering based on the particle's cell ID (Sorted); and a pseudo-random ordering (Random).

As discussed in Section 4, such orderings are expected to have a significant impact on the performance of both kernels, and this is reflected in the GPU results. On the CPU, however, we see very little effect. This is because our kernels do not make use of vector instructions, and thus the stencil operation is not a gather on this platform – the biggest bottleneck for the scalar implementation is cache performance. The size of the electric and magnetic field data is small (12 MB) in comparison to the size of the particle data (1.5 GB), but accessed more frequently, and we therefore see good cache behaviour regardless of particle ordering. Also of note is that the impact of particle ordering decreases with each generation of the CUDA architecture – we attribute this to the introduction (and improvement) of hardware caches.

Since current accumulation is so much more expensive than field accumulation, we use the random particle ordering henceforth – taking a 2× hit on the latter has a smaller effect on overall runtime than a 4× hit on the former.

5.3 Hardware Comparison

The graph in Figure 5 compares the runtime of the final kernel configuration (atomic adds, with a random particle ordering) on each platform. In addition to the performance



(a) Field calculation kernel.

(b) Current accumulation kernel.

Figure 4: Impact of sorting schemes on kernel runtimes.

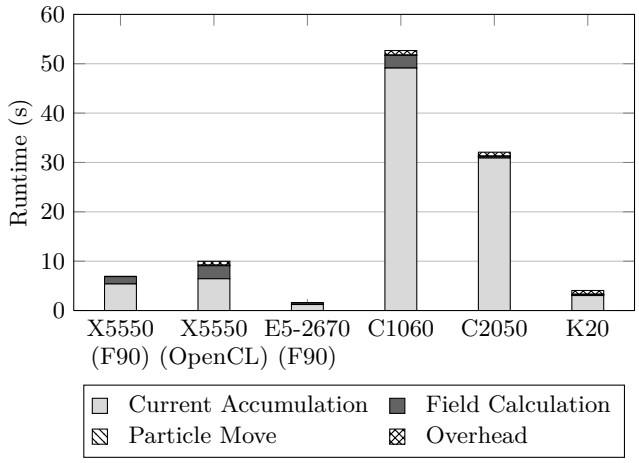


Figure 5: Best kernel performance across platforms.

of each individual kernel, we also list any overheads (*e.g.* PCIe data transfers), to facilitate a fairer comparison between architectures.

The performance of our OpenCL code on the X5550 almost matches that of the native Fortran code. The difference in performance is caused not by the current accumulation kernel (as might be expected) but by the field calculation kernel. We believe this to be due to the change in data layout (from AoS to SoA), which spreads the data for a given particle over several cache lines.

That the performance difference is so low is a positive result for the maintainers of EPOCH3D, suggesting OpenCL will provide a good route to explore accelerator performance without a significant negative impact on CPUs.

What is most clear from our results is that the current accumulation kernel is an issue, dominating the runtimes of all implementations on all hardware. The other kernels perform well, and in line with our expectations. However, overhead is much larger than running the kernels themselves – hiding PCIe communication costs via pipelining, or making more kernels resident on the device, will clearly be a critical direction for future work following the acceleration of the current accumulation step.

Something we do not consider here is how the performance of EPOCH3D on different hardware changes with the number of particles per cell. This impacts simulation accuracy, and current simulations are bounded by performance – running more particles per cell is desirable, but too expensive. It may be that GPU implementations only begin to offer greater performance for these larger, more complicated, simulations; we leave this investigation to future work.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we report on the development of an OpenCL implementation of EPOCH3D, a production PIC code developed by the University of Warwick. The PIC algorithm is highly parallel, and should map well to accelerator architectures. However, as shown in this study, there are two issues that first need to be addressed in EPOCH3D: the highly serial nature of its legacy Fortran implementation; and the presence of write-conflicts in its current accumulation step.

Despite the promising initial results demonstrated in this work, our findings confirm that the current accumulation step (and in particular, its use of atomics) is our biggest barrier to both performance and portability – suggesting that a fundamental change to the algorithm is necessary to fully utilise the massive levels of parallelism supported by emerging parallel architectures.

In future work, we intend to continue our porting effort, and apply the knowledge we have gained during this process as part of ongoing efforts to optimise EPOCH3D for x86-based hardware (including Intel Xeon Phi coprocessors). Additionally, we are developing a mini-app [8] representation of EPOCH3D, in order to facilitate and accelerate the investigation of alternative hardware and software options for PIC codes.

Acknowledgements

This research is supported in part by the EPSRC grant "A Radiation Hydrodynamic ALE Code for Laser Fusion Energy" (EP/I029117/1) and by The Royal Society through their Industry Fellowship Scheme (IF090020/AM).

7. REFERENCES

[1] T. Arber et al. EPOCH: Extendable PIC Open Collaboration. `http://ccpforge.cse.rl.ac.uk/gf/project/epoch/`, October 2011.

[2] N. Arora, A. Shringarpure, and R. W. Vuduc. Direct N-body Kernels for Multicore Platforms. In *Proceedings of the International Conference on Parallel Processing*, ICPP '09, pages 379–387, Vienna, Austria, September 2009. IEEE Computer Society.

[3] R. Bordawekar, U. Bondhugula, and R. Rao. Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application! Technical Report RC24982, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, 2010.

[4] R. Bordawekar, U. Bondhugula, and R. Rao. Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. Technical Report RC25033, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, 2010.

[5] H. Burau et al. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *IEEE Transactions on Plasma Science*, 38(10):2831–2839, 2010.

[6] J. Dongarra et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[7] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-platform GPU Programming. Technical Report UT-CS-10-656, Knoxville, TN, 2010.

[8] M. A. Heroux et al. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, Albuquerque, NM, 2009.

[9] R. G. Joseph, G. Ravunnikutty, S. Ranka, E. D'Azevedo, and S. Klasky. Efficient GPU Implementation for Particle in Cell Algorithm. In *Proceedings of the IEEE International Parallel Distributed Processing Symposium*, IPDPS'11, pages 395–406, 2011.

[10] Khronos OpenCL Working Group. OpenCL 1.2 Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[11] M. Klemm et al. Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures. In *Proceedings of the International Workshop on OpenMP*, IWOMP '12, pages 59–72, Rome, Italy, 2012. Springer-Verlag.

[12] K. Komatsu et al. Evaluating Performance and Portability of OpenCL Programs. In *Proceedings of the International Workshop on Automatic Performance Tuning*, iWAPT '11, Berkeley, CA, June 2010. Springer.

[13] V. W. Lee et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pages 451–460, 2010.

[14] P. Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[15] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, New Orleans, LA, November 2010. IEEE Computer Society.

[16] OpenACC Corporation. OpenACC 1.0 Specification. `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`, November 2011.

[17] OpenMP Architecture Review Board. OpenMP 4.0 Specification. `http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf`, November 2012.

[18] OpenMP Architecture Review Board. Technical Report on Directives for Attached Accelerators. Technical Report TR1, November 2012.

[19] S. J. Pennycook et al. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing*, (to appear), 2012.

[20] S. J. Pennycook and S. A. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. In *Proceedings of the International Workshop on Performance Modeling, Benchmark and Simulation of HPC Systems*, PMBS '12, Salt Lake City, UT, November 2012.

[21] Ruhl, H. Classical Particle Simulations with the PSC Code. `http://www.physik.uni-muenchen.de/lehre/vorlesungen/wise_09_10/tvi_mas_compphys/vorlesung/Lecturescript.pdf`.

[22] M. Smelyanskiy et al. Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1563–1570, November 2009.

[23] G. Stantchev, W. Dorland, and N. Gumerov. Fast Parallel Particle-To-Grid Interpolation for Plasma PIC Simulations on the GPU. *Journal of Parallel and Distributed Computing*, 68(10):1339 – 1349, 2008.

[24] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. E. Guney, and A. Shringarpure. On the Limits of GPU Acceleration. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2010.

[25] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Optimization of a Lattice Boltzmann Computation on State-of-the-Art Multicore Platforms. *Journal of Parallel and Distributed Computing*, 69(9):762–777, September 2009.